Internals of VORTEX: The Source Editor*

John Coker
Computer Science Division,
University of California,
Berkeley, CA 94720

October 4, 1988

1 Introduction

VORTEX[2] is a source-based interactive document preparation, system built on the TEX typesetting language[4], which allows some of the functions of direct-manipulation systems as well. This report describes the editor portion of the prototype system as currently implemented.

VORTEX is an interactive system based on TEX. TEX allows finer control and produces higher quality typesetting than the other systems commonly available on UNIX. VORTEX makes these features more accessible through integration with a powerful editor, an incremental TEX processor and a what-you-see-is-what-you-get output displayer.

The first version of VORTEX is nearly finished and has shown us what problems our early assumptions and the design have produced. This document describes the internals of the system from the point of view of the VORTEX source editor (the user interface). Since VORTEX is written in three separate pieces, this paper does not describe the architecture of the entire system. However, it does document most of the interfaces within the system.

The references at the end of this report can be followed up for more information, especially the Ph.D. thesis of Pehong Chen[1]. More information on the internal represenation can be found in [3].

*Sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command, under Contract No. N00039-88-C-0292.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1988		**00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Internals of VORTEX: The Source Editor	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**19**	

1.1 Some Associations

The system runs under X version 10, and appears to be another of the EMACS[5] family of visual editors. The command structure and invocation is based on the EMACS paradigm. The source editing paradigm is also that of EMACS with point (the left edge of the text cursor) and mark (an invisible text marker) used to delimit text.

VORTEX is actually implemented as several processes. The one the user starts up (which is called `vortex`) implements the source editor. The source editor contains a Lisp interpreter which is used to invoke the other processes, although this is typically done by the system when needed. From the user's point of view, he interacts solely with the source editor since its Lisp interpreter performs all interpretation and executes commands at the higher levels.

1.2 A Few Words About the Editor

The program `vortex` is a text editor with a built-in Lisp interpreter which has been written in C to perform all the user interaction for the VORTEX system. It runs only under X, since is windows (in EMACS terminology) are also X windows. Every different "locus of editing" is directed through a separate window. In most instances, each window views a different source file or page of the proof output, although it is quite possible to edit the same file and view the same page at the same time in different windows.

Our Lisp interpreter, vLisp, has been built into the editor (it kernel is entirely written in C) and all the editor functions are accessed through the Lisp system. Higher level editing is done through vLisp functions. The Lisp kernel and most primitive editing functions are coded in C and almost all of the visible system is built up from these primitives through vLisp functions.

2 The Editing Paradigm

In order to implement the functionality required, we decided to use an "internal representation" which was equivalent neither to the source text of a TEX document nor the primitive typesetting commands of a `dvi` file. We needed something that could represent both, and ultimately be mapped back into the TEX source. However, due to the lack of structure of TEX source, this was impractical although we did manage to achieve approximately the same functionality.

2.1 Our Problems with TEX

The largest problem is that there is very little underlying structure to TEX. There is a form of scoping (groups, delimited by { and }). And one can imagine words and symbols

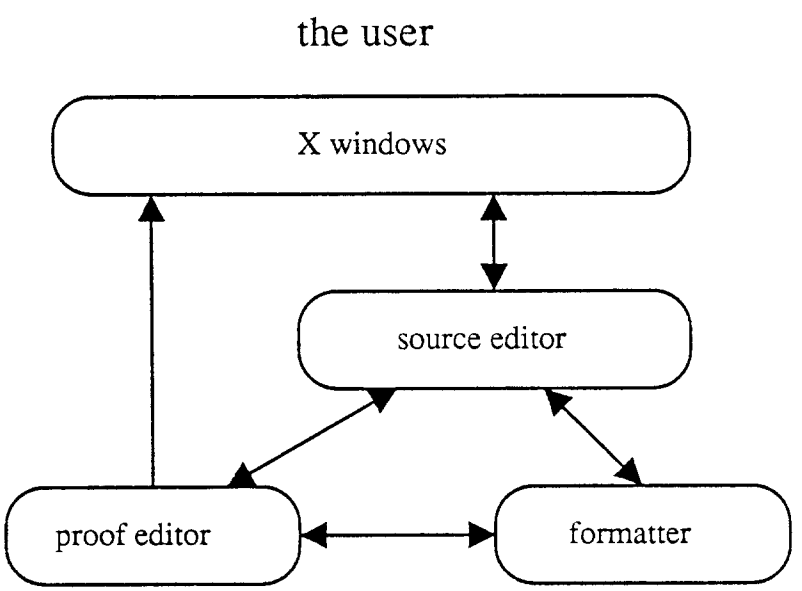

Figure 1: The VORTEX System Layout

as children of paragraphs, but the scoping is not bound into the logical structure and there is no control over side-effects. All control structures and user definitions are done with macros, which may be changed at any time. In fact, a TEX program may change the reader syntax conditionally.

Thus, it is not practical to get completely away from the text-based structure. We could not design an internal representation that would allow us to generate correct and reasonable TEX source from some abstract representation of a document. Thus, in the implementation, we ended up with an internal representation which is used to map between the basic representations of the document, the "source" (TEX code) and the "proof" (e.g., `dvi` command).

The terminology used denotes the tree-based internal representation as the IR_I and the source text representation (basically ASCII text) as the IR_S and the target representation (images to print or display) as the IR_T. Basically, the "internal representation" is a tree with the interior nodes representing logical structure (pages, paragraphs and words) and the leaves pointing to the TEX source which implements it. The tree also has links (filled in as necessary) into the IR_T at all levels.

Thus, we end with a system which appears as through it has a fundamental representation, but is actually a way to map the logical and display elements back into the source stream. We cannot make a change in the proof editor and fix up the text, we must use the IR_I to map the change back into the source and then re-format the portion of the document which has been changed. This then fixes the IR_I and IR_T.

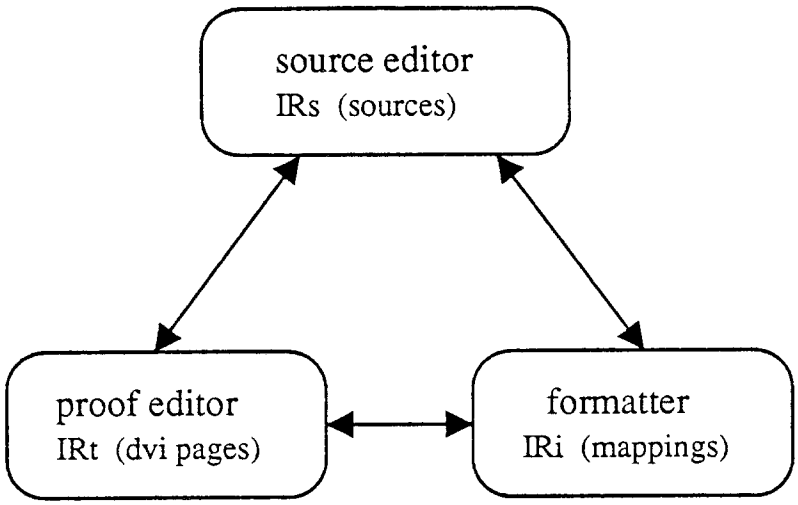

Figure 2: Locations of the Internal State

2.2 What We Have Done

We make it appear as through the user can edit either in the source using EMACS commands or on an image of the formatted document using WYSIWYG[1]-style commands. The source updates instantly and the representations are kept synchronized. When editing on the proof, it synchronizes after each editing operation. When editing the source, merely typing space synchronizes the proof representation.

2.3 Separation of Responsibility

The source editor is responsible for maintaining the most basic representation of the document, the TEX program text. Since VORTEX operations on the typical UNIX paradigm (read in files, edit and write them back), the only representation stored externally is the source text. During its execution the formatter will have a current copy of the source, its own IR_I and information on the IR_T. The proof editor will also have it's own cache of the IR_T (page-level granularity) and other command-state and selection information.

The source editor program, `vortex`, can run with or without the other two processes. In fact, starting up VorTeX does not automatically start those daemon programs. The user may explicitly start the formatter with `start-formatter` or the proof editor with `start-proof-editor` or implicitly with `format-document` or `proof-document` respectively. `make-document` starts up the formatter, transmits the entire contents of the TEX document (possibly more than one file) and begins treating the source buffer(s) as part of the WYSIWYG system.

[1]WYSIWYG–What You See Is What You Get.

Editing in a buffer which is part of the formatted document causes the source editor to send the changes to the formatter. However, due to the possibility of syntactic inconsistencies, the formatter will not reorganize the IR_I until the user explicitly requests it with `format-document`. The command `format-document` will only send the changed characters from the document's source buffers and trigger an incremental re-format. The user is expected to format the document often, whenever he wishes to see the formatted version. In fact, a version of the system could be built which would request formatting every time the user made a change to the source (although we think this would be prohibitively slow).

To understand the following descriptions one must keep in mind the fact that the system only truly maintains the source text of the document. The formatter translates this into the proof version with which VORTEX can display the formatted document and translates what little structure there is in the formatted representation back into the source text stream.

2.4 The Selection is the Thing

All proof editing works based on the "current selection." In the EMACS paradigm, the current region is defined by the point and most recent mark. In the proof editor, the selection is defined by selecting one or more pieces of the document at some level of structure. Assuming that these all come from some ordered, contiguous portion of the source document, the proof selection can be mapped back into the source text as an EMACS region.

Making a selection in the proof editor, causes it to send the selection back to the source editor. The proof selection is available to Lisp programs through the command `proof-selected-region`, which returns the current proof selection as an EMACS region (plus the buffer name). The Lisp functions which implement the WYSIWYG commands use this mechanism to set up the current buffer, point and mark and then use EMACS editing commands to make the change.

The proof selection can also be set from the source editor by translating point and mark into a list of characters for the proof editor to display. This mechanism is not used much, but the trivial case, selecting a single character, is very useful for finding the current position in the other representation (`C-x .` is bound, in both editors, to a function which selects the current point in the other editor).

This has the advantage of being easy to extend, since writing a Lisp function is all that is required, although the lack of any true knowledge of the document makes more sophisticated transformations impractical.

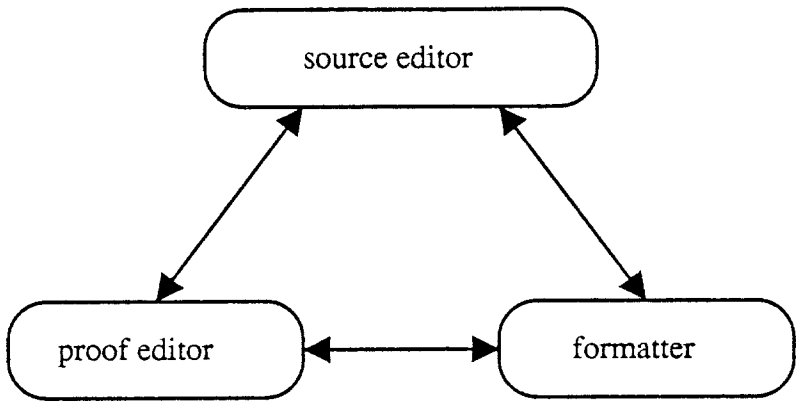

Figure 3: The Basic System Triangle

2.5 The Mapping Mechanism

The proof editor maintains a copy of the IR_T which associates TEX boxes with dvi-style output primitives. It can paint a window with the glyphs and rules to display the formatted output and translate the mouse cursor position into a formatter box ID. These IR_I box IDs may represent terminals (characters), called t-boxes, or logical structure (words, paragraphs and pages), called n-boxes.

Each character in the IR_S is tagged with a unique ID which allows the source editor to translate t-box IDs as sent from the proof editor into source buffer positions which can be used for editing. Thus, when the proof editor sends the selection as a pair of t-box IDs, the source editor requests the corresponding buffer/region from the formatter and sets up the data returned by `proof-selected-region`.

3 The Three Components

The three basic pieces to the VORTEX system are the *source editor*, *proof editor* and *formatter*. These have been implemented as separate processes which communicate through Berkeley UNIX TCP/IP, "stream sockets." For the most part, the communications are asynchronous, without acknowlegements and avoiding the necessity for request/reply messages. However, there are several places where queries were unavoidable. The source editor is the program run by the user to invoke the system, it then invokes the processes which implement the proof editor and formatter as daemons.

Along each of the three communications channels editor, *iTEX*–the TEX formatter and *source*–the source editor) a constant-length header has been defined and message-specific data trails the header. Other than the data length parameter in the header, there are no message boundaries.

3.1 The Source Editor

The source editor invokes the proof editor and formatter in response to the commands `start-proof-editor` and `start-formatter`. The formatter is also started, if necessary, when the user creates a new TEX document with `make-document`. Similarly, the proof editor is started automatically by `proof-document`.

Since `vortex` can function as an editor with both, one or none of the daemons running, it does not exit if they do. The user may get rid of the proof editor and formatter with `kill-proof-editor` and `kill-formatter` respectively. However, without the two daemons, only normal text editing may be done.

Note that all real editing is done in the source editor. The proof editor displays the formatted output and allows selection on that output, but the system translates all selections back into the source buffers and is edited there.

All files handled by the system are managed by the source editor. This is necessary since the method of invocation of the formatter allows it to be run on a different host machine.

3.2 The Proof Editor

The proof editor appears to be identical to the source editor in terms of paradigm–it's windows operate as do source buffer windows. Any VORTEX buffer window may visit a proof buffer; the differences between a source and a proof window are all implemented at the buffer level.

The proof editor commands are implemented as a small set of remote procedure calls to the proof editor daemon, which is responsible for maintaining the image in the window. The mode line and window itself is managed by the source editor and all input is filtered through the key bindings scheme and devolves into Lisp function calls.

3.3 The Formatter

The formatter is not accessible by the user directly. It is responsible for formatting the document and performing translation functions in both directions, but the user never needs to interact with it. As with the proof editor, all functionality is implemented through a set of remote procedure calls.

The top-level structure is the *document*, which contains a set of files. Every character in the system belongs to a file and has a unique character ID assigned it by the source editor. The character ID encodes the file ID within it. Once formatted, that character has a direct relationship with a t-box, the representation of the character in the formatted page.

When a file needs to be sent to the formatter, the source editor makes sure it exists in a buffer, marks the buffer as a special TEX file buffer, and sends the contents of the file to the formatter. These file transfers are always initiated by the formatter when it requires a new file. When the source editor needs to update the contents of a file, the changes are sent in terms of *insert* and *delete* remote procedure calls.

When a file marked as a TEX file buffer is changed (inserted into or deleted from) the source editor must send the changes to the formatter before any other processing (using either of the daemons) is done. Currently, the lowest-level insert and delete commands in the source editor translate into formatter remote procedure calls immediately. Thus, formatting occurs during normal text editing, and an explicit request to format the document may find it already in a consistent state.

4 System Lisp Commands

All low-level control of the two "daemon" process is done through Lisp commands most of which correspond to the IPC requests described in the next section. Since the source editor controls the formatter and the proof editor, but does not participate in their communications (the formatter-proof editor link), we have no Lisp commands which control that edge of the triangle.

4.1 Proof Editor Commands

The proof editor presents the same paradigm as the source editor–its windows operate as do source buffer windows. Any VORTEX buffer window may visit a proof buffer; the differences between a source and a proof window are all implemented at the buffer level. The commands listed in this section are the lowest level of all (they generally translate directly into low-level communications operations). The majority of editing functions are more complex functions written in vLisp and invoked through key strokes in the normal EMACS manner.

`start-proof-editor`
starts up the proof editor on the specified host if necessary and connects to it. The function returns `t` if a sucessful connection was made and nil otherwise. If there is a serious low-level communications failure, an error may occur.

The proof editor is run on the machine given by the value of `proof-editor-host` by exec'ing the program given by `proof-editor-program` on that host (`"localhost"` by default).

If the proof editor is already running when this function is called, the current proof editor is killed (as with `kill-proof-editor`) and a new one is started.

`kill-proof-editor`
kills the proof editor if one is running. If a formatter is running, it loses its connection to the proof editor also.

`proof-document`
creates a proof window on the given document at the specified physical page number, or at page one if none is specified.

`proof-goto-page`
changes the page being viewed in the current window (or the window specified by the second argument if there is one) to the specified physical page number.

The physical page number has nothing to do with the page number TEX prints on the bottom of each page, it refers to the order of the pages as laid out in the document. The first page formatter is page one, the second page two.

If the specified page does not exist, the editor finds the closest possible page. Thus, both zero and one go to the first page and numbers greater than the length of the document go to the last page.

`proof-next-page`
changes the page being viewed in the current window (or the window specified by the second argument if there is one) forward or backward by the specified count.

`proof-move-absolute`
moves the current window (or the window specified by the second argument if there is one) to the given proof editor position. A proof buffer window position is specified in units of screen pixels. These routines are normally not needed by the user.

`proof-move-relative`
moves the current window (or the window specified by the second argument if there is one) in the given proof editor relative to the current position.

`proof-select`
causes the area under or the current mouse position (it must be invoked interactively) to be selected at the next higher level of structure.

This function and the other selection functions below, cause VORTEX to translate the selected structure in the proof window into a source buffer/region pair as returned from `proof-selected-region`. Proof selections are performed by the proof editor, with the reverse mapping done by the source editor upon completion.

`proof-select-more`
causes the area under the mouse cursor (it must be invoked interactively) and all the text between it and the current selection (begun with `proof-select`) to become the current selection. It always operates at the granularity established by `proof-select`. Thus, if the user has clicked the left button twice at the same place (calling `proof-select` to select a word), and then moves to another word and clicks the middle button (calling `proof-select-more`), all words between and including the ones clicked on will be

selected

`proof-selected-region`
using the proof editor as a list. The first element in the list is the source buffer name (the general buffer handle) and the other two are offsets into the buffer defining the region, in order.

If no region has been selected with the proof editor, this function returns nil.

`proof-moveto`
scrolls the specified proof window to the given offset in the buffer. If no buffer and window are specified, the last used source and proof window buffers are used.

4.2 Formatter Commands

The formatter does not appear to the user directly. It is responsible for formatting the document and performing translation functions in both directions, but the user never needs to interact with it. As with the proof editor, all functionality is implemented through a set of remote procedure calls.

`start-formatter`
starts up the formatter on the specified host if necessary and connects to it. The function returns t if a sucessful connection was made and nil otherwise. If there is a serious low-level communications failure, an error may occur.

The formatter is run on the machine given by the value of `formatter-host` by exec'ing the program given by `formatter-program` on that host (`"localhost"` by default). If we're already connected to the proof editor and a sucessful connection is made to the formatter, a connection between the formatter and the proof editor is established

`kill-formatter`
kills the formatter if one is running. If a proof editor is running, it loses its connection to the formatter also (but doesn't necessarily die itself).

`make-document`
creates a new document whose master file is that specified by the given TEX source file. If the file is not being visited in any buffer, it is visited. Buffers which are visiting TEX source files are marked as such so that changes to them can be communicated to the remote TEX formatter. The user will not allowed to kill this buffer until the document is closed.

The contents of this master file buffer will be scanned by the formatter and processed as TEX source code. Note that changes to the source buffer are sent to the formatter program when (or before) the buffer is written or when a `format-document` command is invoked.

Since only one document is allowed at a time in the current system, `make-document` will implicitly call `close-document` if there is a current document when it starts. To

reformat the current document, use `format-document`.

If a formatter process is not running when this function is called, one is started. `make-document` implicitly calls `start-formatter` in this case. See the documentation on the latter function for more information.

The contents of the buffer are sent immediately. If other files are required (via a `bsl input` statement), they will be sent as requested by the formatter.

`format-document`
sends a message to the incremental TEX process to begin reformatting the current document. There must be a current document previously opened with `make-document`.

`close-document`
closes down the formatter connection, which has the effect of freeing all resources used by the document.

5 Communications Protocols

Along each edge of the process triangle, a separate communications protocol has been defined. The source editor-formatter and source editor-proof editor connections are described in detail here. Each of the three protocols include a global protocol, which implements rendezvous and common functionality.

All numbers are transmitted in network byte order. Two sizes are used: **long** (32 bytes) and **short** (16 bytes) either signed or unsigned. Other data (ASCII strings) is sent as a byte stream. There are no padding requirements for messages, although we have been careful to lay out the data within each message to avoid architecture differences so that the data can be easily read into a C structure or array. Strings are not necessarily terminated with a NUL character (ASCII 0).

Each packet begins with a constant format header. The eight bytes of header may make up the entire message. These headers are constant, however the trailing data is defined by each request.

u_short	request	the request code
u_short	datalen	the trailing data length (bytes)
u_long	commdata	communications specific data

The "communications specific data" is not used in the global protocol, but stores the window ID and file ID for the proof editor and formatter communications respectively. These IDs allow the two processes to pass the most common piece of information without needing to define additional protocol.

5.1 Global Protocol

The basic format of all messages and the messages common to all three edges of the triangle are listed in the include file "gl_comm.h". A copy of this file and the other three message sets are appended to this report. See those files for a more concise description of the byte layout of the individual messages.

5.1.1 Process Rendezvous

Each of the two daemon processes needs to connect to the source editor and to each other. Upon start-up, each daemon examines its argument list to determine how to call the source editor back. Two non-option arguments are expected, the host name and internet port number on which the source editor is listening. In addition, the proof editor takes the X display name as a third argument. (Both programs should also allow options to be passed in, particularly the `-d` option for debugging.)

Once a daemon has successfully connected, it should send a `VERIFY` request on the new connection. The source editor will respond with a `WELCOME` or `GOAWAY`. The `VERIFY` packet contains an identifier (a "magic number") and the protocol versions of the global and specific local communications. Assuming the identifier is correct and the versions match, a `WELCOME` is sent with no data, otherwise it responds with a `GOAWAY` message and the data contains an error message (text string). Once the `WELCOME` message has been received, the daemon is officially connected to the source editor and no more handshaking is necessary.

5.1.2 Connecting the Formatter and Proof Editor

There is one other rendezvous task that must be done; connecting the formatter and proof editor to one another. The protocol we've implemented to perform this is necessary because of the possibility of running more than one VORTEX on a single machine; we could not just assign the internet port numbers.

The protocol used for this rendezvous involves a three step process: 1) the source editor instructs one daemon to listen for a connection from the other with a `LISTENAT` request, 2) the daemon responds with a `LISTENING` reply when it is ready, and 3) the source editor instructs the other daemon to connect to the first one with a `CONNECT` request.

The `LISTENAT` request includes an internet port number and a time-out value. However, the port number it specifies need not be the one used by the daemon. It must attempt to bind internet ports starting with the number specified and incrementing until the operation succeeds (or fails with some error other than "port already in use"). Once it has successfully bound the port, it returns the port number to the source editor and waits for at most the given time-out for a connection from the other daemon.

Once the two daemons have established the TCP/IP connection, all of the system is in proper communication. Whenever the second of the two daemons successfully connects back to it, the source editor attempts to establish a connection between the daemons as described above.[2]

5.1.3 Error Handling

There are two main jobs involved in error handling; reporting and re-synchronization. The first is obvious, but the second requires some throughout. Since we've implemented the VORTEX protocol on TCP/IP streams (which have no message boundaries), we need some method of marking the stream for re-synchronization.

This resynchronization is done with the "out of band data" facility of UNIX sockets. The `FLUSH` request sets a mark (using an out of band character) and the receiver flushes all bytes up to that mark. In practice it has not happened that the programs lost their position in the byte stream (since the headers are regular), but it may be necessary at some time in the future.

There is an additional request, `ERROR`, which sends an ASCII error message destined for the user. These requests are really only used to send errors back to the user to be displayed by the source editor. `ERROR` is a request just like any other and does not imply `FLUSH` or `ABORT`.

There are two messages which terminate the connections (and the daemon processes when sent from the source editor). `QUIT` signals termination (usually due to a user command) of a daemon and `ABORT` is considered an error termination. Both have the same semantics (no trailing data) and are handled the same way by the current source editor. When sent from the source editor, they command the specified daemon to quit and when sent from a daemon, signal that the daemon is quitting. The daemons cannot command the source editor to terminate.

5.2 Proof/Source Protocol

The Lisp functions which implement proof editor functionality are all executed by the source editor and presumably result in calls to the lowest-level proof operations. In addition to these, there are many implicit operations which are not visible to the user such as *create window*, *expose window* and *destroy window*.[3] See the include source file `"ps_comm.h"` for the exact format of all the proof editor procedure calls.

In all proof/source requests which require one, the *window ID* of the proof window in question is passed in the `commdata` field of the message header (bytes 4 through 8). The ID passed is actually the X window ID of the proof window (which corresponds to

[2]The Lisp programmer may also explicitly establish the connection with `make-connection`.

[3]These implicit requests mirror the X version 10 window management paradigm.

a proof window and buffer for the source editor). For all requests other than CREATE any system of unique identifiers would have sufficed, but we chose the X window ID for convenience.

5.2.1 Window Management

The first of the requests in this link of the system implement window management functions. The most obvious place to start is window creation. The request CREATE sends the information on a newly made X window which the source editor is using to display the particular buffer. The window has already been created and mapped and this request includes an implicit EXPOSE of its entire surface.

u_short	Xoffset	X offset on window
u_short	Yoffset	Y offset on window
u_short	width	width of valid portion of window
u_short	height	height of valid portion of window
u_long	fg_pixel	pixel value of foreground
u_long	fg_pixmap	tile for foreground
u_long	bg_pixel	pixel value of background
u_long	bg_pixmap	tile for background
u_long	hl_pixel	pixel value for high-light
u_long	hl_pixmap	tile for high-lighting

The first four arguments define the region on the X window which is owned by the proof editor. The region (X, Y to X+width, Y+height) is the only area on the X window which may be painted by the proof editor. The proof editor is not allowed to change the window in any way other than repainting its designated region. It may not even use XClear to erase the window since the title bar is drawn on the same X window by the source editor.

The last six arguments are used as the X color handles (more IDs) for painting. These are specified individually for each window since it may be useful for different proof windows to have different colors schemes. The comments given after the element definitions should be enough explanation to the X programmer.

The obvious accompaniment to create is DESTROY, which is sent with no trailing data. A window may only be destroyed by the source editor and no X access may be made to the window after the request has been sent (and the editor once more reaches the top level). All proof editor data structures used for this window should be discarded although cached page data should be expired by some other means for efficiency when the user opens another window.

The two remaining functions for window management are RESIZE and EXPOSE. These both pass a rectangle although the rectangle has different meanings in each. The rectangle in RESIZE is just like that in CREATE; it re-defines the drawing area on a particular window. For EXPOSE the rectangle is offset from the offset specified by the most recent RESIZE or the CREATE (e.g., an EXPOSE of the entire window would result in 0, 0 for the exposed area's origin). The area should always be within the defined drawing area.

5.2.2 Operation Batching

A less important enhancement to the protocol is the notion of batching requests. The source editor will surround a set of requests which form a single user command with a BSTART/BEND pair. This allows the proof editor to maintain state about flushing the X output queue or improving the "feel" of the system by making the single conceptual operation appear to occur all at once.

These batching groups may nest (for the sake of generality) although groups deeper than one level will have the same significance as one. The current source editor never uses more than one group at a time.

Each batch level applies to the window specified in the packet header only, although in the current implementation no other messages will appear during the batch grouping.

5.2.3 Document Commands

Each window has a notion of the current document (as well as page and position) and the request DOCUMENT requests that a different document be displayed in the specified window. The proof editor then uses the document ID given as the trailing data to query the formatter for the page's formatted version.

The proof editor implicitly displays the document, displaying the upper-left hand corner of the first (physical) page. In the current VORTEX, the formatter can only handle one document, so any instance of the proof editor will always be called with a single document ID for all windows. As soon as the global document information has been successfully gotten from the formatter, the proof editor must send the source editor a DOCPAGES request to inform it of the number of formatted pages in the document.

The source editor requests to move between pages fall into two major categories: physical and logical page specification. Physical pages are those established by the order in which TEX writes pages into a dvi file. To move between physical pages in a document, two requests are used: GOTOABS and GOTOREL. GOTOABS takes the physical page number and moves to that page, numbered from one through the number of printed pages. GOTOREL just moves forward the specified number or pages (negative numbers move backward).

Logical page specifications are defined by the numbers stored into the ten \count

registers. The \count0 register is used to store the number to be printed at the bottom of the page (\pageno is a synonym for \count0). The other count registers are used by various macro packages for other formatting counters. Since there are ten count registers, a logical page specification may have as many as 10 components, which match the numbers in the \counts explicitly (as a number or range) or implicitly by being omitted. See the user document for the exact format of this page specification (the source editor never needs to generate these specifications itself–they always come from the user directly). The LOGICAL request passes a string to the proof editor for interpretation.

The other layer of movement commands are those to move the page being displayed in the window under the visible portion of the editor window. There are two of these commands: MOVEABS and MOVEREL. Both commands take an X and Y distance in pixels and cause motion of the specified amount (for MOVEREL, negative X and Y mean left and up respectively).

There is one last positioning command, although currently it is the only one of a one of a number of similar function. The POSITION request specifies an n-box in the document being displayed and merely instructs the proof editor to "make that box visible." This usually entails moving to another page or moving on the current page, but is a very unstructured command and is one of the major places where the proof editor defines the feel of the system.

5.2.4 Manipulating the Proof Selection

"The proof selection" is a concept which has a simple meaning to the user and a complex one within the system. In general, such a selection is a list of n-boxes which happen to make up some logical piece of structure on the page. However, in the current implementation, the current selection is a range of characters in the source buffer which limits it to contiguous pieces of text which do not overlap document structure (letters, words and paragraphs).

The selection is established (or re-established) with the SELECT request from the source editor. The source editor specifies the position of the mouse cursor and the proof editor must decide what piece of the document this signifies. To define a (new) selection, the X and Y position of the mouse within the drawing area of the window is sent. To un-define the selection a SELECT with no trailing data is sent.

Once a selection has been defined, it may be expanded in two ways, either by moving to a higher level of document structure or by selecting a range of objects at the current level. When the proof editor receives a SELECT at the same mouse position as the last, it moves up a level (from a character to a word to a paragraph). When the mouse position changes, a new selection is made, starting again at the character level. Expanding a selection which involves more structure at the same level as the current selection is done by sending a SELECTMORE request with a new mouse position. All the pieces of structure

from the old selection through the unit defined by the new mouse position become the current selection.

Whenever a selection is established or expanded, the proof editor must return the first and last *character* in the selection using a `SELECTION` request, with the first and last t-box IDs as the trailing data. If there was an error, or the selection is undefined for some reason, a `SELECTION` request with no trailing data is returned.

5.3 Formatter/Source Protocol

The Lisp functions which implement formatter commands are executed by the source editor and result in remote procedure calls to the formatter. See the include file `"ts_comm.h"` for the exact format of the formatter remote procedure calls.

In all formatter/source requests which require one, the *file ID* of the affected source file is passed in the `commdata` field of the message header (bytes 4 through 8). The ID passed is constructed by the source editor and can only range from 1 to 127. These file IDs are assigned by the source editor at the time a TEX buffer is first dealt with and the file ID is encoded into each character ID it contains.

5.3.1 Document Handling

Currently, the VORTEX formatter can only handle one document, so the source editor closes down the formatter connection when a document is closed and creates a new one when a new document is created. Because of this, there is no explicit request to create a new document. The first `FORMAT` request contains an implicit document creation command. The `FORMAT` request contains the name of the master file (and the corresponding file ID in the packet header). This establishes the correspondence between the *master* or *root* file of the document and its file ID.

When the source editor creates a new document, it justs asks the formatter to format the document. The formatter then queries for the contents of the file with a `TEXINPUT` request, which specifies the file name in the trailing data. A `TEXINPUT` is also sent when the formatter encounters an `\input` command in the TEX source. Assuming the file can be found, the source editor replies with a `OPENFILE` request which contains the contents of the source file.

The file IDs and character IDs within those files are assigned by the source editor at the time the file is loaded into the system. Each character is represented by four bytes (the character ID) internally. This code contains the file ID, the character code and a "unique number" for that character code over all files.

In the present system, the source editor is the only piece that directly changes the contents of the document. This is done (as the user edits a buffer) by sending `INSERT` and `DELETE` requests to the formatter. The `DELETE` request just specifies a range of characters

(in the source buffer) to be deleted. The INSERT request passes a position and a list of characters to be inserted before that position.

5.3.2 The Mapping Facility

The source editor is constantly translating box IDs and buffer/offset values to perform the mapping functions. These requests are implemented as the only two cases of a more general facility, EXECUTE. An EXECUTE request specifies a sub-request code and any trailing data it requires and expects to receive a RETURN reply. The RETURN contains the results of the EXECUTE or no data at all on error. The two functions used for the mapping are TGT2SRC and SRC2TGT. The "source" representation of a character is the file ID and offset within that file. The "target" representation is the t-box ID used in the formatter/proof editor communications.[4]

To implement the function "scroll proof window to point," the source editor queries the formatter for the t-box which corresponds to the text cursor in the source (using SRC2TGT) and issues a POSITION command to the proof editor with the resulting box ID.

To implement the proof selection in terms of the source editor model, the first and last t-box IDs are translated into source buffer positions (using TGT2SRC). Assuming the files are the same and the positions in order, it can then assign the current point and mark to the region defined by the proof selection.

6 Restrictions and Assumptions

The packet header contains just enough information to allow the serialization and deserialization routines to be independent of the code which calls or handles the individual requests. Each request specifies the format of the data (if any) individually.

Since there is only one message length in the header, we never use more than one variable length field, although it would still be possible to do so (with a second length in the data or a terminator convention). Variable length data is avoided whenever possible.

The programs assume that the packet header is written in one unit (one call to **write**) and may fail if the header is written in pieces. Thus, the header should be written out as one chunk and the data in whatever form is most appropriate. The source editor always writes the header in one system call and the data in another (always two calls to **write** if the packet contains data). (This could also be achieved with the single system call **writev**.)

[4]This is **not** the same as the character ID assigned by the source editor.

References

[1] Pehong Chen. *A Multiple Representation Paradigm for Document Development.* PhD thesis, 1988.

[2] Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VORTEX document preparation environment. pages 32–24, June 19–21 1986.

[3] Pehong Chen and Michael A. Harrison. Integrating noninteractive document processors into an interactive environment. Technical Report 87/349, April 1987. Submitted for publication.

[4] Donald E. Knuth. *The TEX Book.* 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[5] Richard M. Stallman. EMACS: The extensible, customizable self-documenting display editor. pages 147–156, June 8–10 1981. A somewhat extended version appears in *Interactive Programming Environments*, Barstow et al. (eds.), McGraw-Hill Book Company, 1984, pp. 300–325.